CHAPTER 3

Regular Expressions and Math with Python

In this chapter, we discuss two modules in Python: *re*, which contains functions that can be applied for regular expressions, and *SymPy*, for solving mathematical problems in algebra, calculus, probability, and set theory. Concepts that we will learn in this chapter, like searching and replacing strings, probability, and plotting graphs, will come in handy for subsequent chapters, where we cover data analysis and statistics.

Regular expressions

A regular expression is a pattern containing both characters (like letters and digits) and metacharacters (like the * and $ symbols). Regular expressions can be used whenever we want to search, replace, or extract data with an identifiable pattern, for example, dates, postal codes, HTML tags, phone numbers, and so on. They can also be used to validate fields like passwords and email addresses, by ensuring that the input from the user is in the correct format.

Steps for solving problems with regular expressions

Support for regular expressions is provided by the *re* module in Python, which can be imported using the following statement:

```
import re
```

If you have not already installed the *re* module, go to the Anaconda Prompt and enter the following command:

```
pip install re
```


G. Rajagopalan, *A Python Data Analyst's Toolkit*, https://doi.org/10.1007/978-1-4842-6399-0_3

Once the module is imported, you need to follow the following steps.

1. **Define and compile the regular expression**: After the re module is imported, we define the regular expression and compile it. The search pattern begins with the prefix "r" followed by the string (search pattern). The "r" prefix, which stands for a raw string, tells the compiler that special characters are to be treated literally and not as escape sequences. Note that this "r" prefix is optional. The compile function compiles the search pattern into a byte code as follows and the search string (and) is passed as an argument to the compile function.

   CODE:

   ```
   search_pattern=re.compile(r'and')
   ```

2. **Locate the search pattern (regular expression) in your string**:

   In the second step, we try to locate this pattern in the string to be searched using the search method. This method is called on the variable (search_pattern) we defined in the previous step.

   CODE:

   ```
   search_pattern.search('Today and tomorrow')
   ```

   Output:

   ```
   <re.Match object; span=(6, 9), match="and">
   ```

A match object is returned since the search pattern ("and") is found in the string ("Today and tomorrow").

Shortcut (combining steps 2 and 3)

The preceding two steps can be combined into a single step, as shown in the following statement:

CODE:

```
re.search('and','Today and tomorrow')
```

Using one line of code, as defined previously, we combine the three steps of defining, compiling, and locating the search pattern in one step.

Further reading: Refer to this document for an introduction to using regular expressions with Python:

https://docs.python.org/3/howto/regex.html#regex-howto

Python functions for regular expressions

We use regular expressions for matching, splitting, and replacing text, and there is a separate function for each of these tasks. Table 3-1 provides a list of all these functions, along with examples of their usage.

Table 3-1. *Functions for Working with Regular Expressions in Python*

Python function	Example

re.findall( ): Searches for *all* possible matches of the regular expression and returns a list of all the matches found in the string.	CODE:
	`re.findall('3','98371234')`
	Output:
	`['3', '3']`

re.search( ): Searches for a single match and returns a match object corresponding to the first match found in the string.	CODE:
	`re.search('3','98371234')`
	Output:
	`<re.Match object; span=(2, 3), match="3">`

re.match( ): This function is similar to the *re.search* function. The limitation of this function is that it returns a match object *only* if the pattern is present *at the beginning* of the string.	CODE: `re.match('3','98371234')` Since the search pattern (3) is not present at the beginning of the string, the match function does not return an object, and we do not see any output.

(*continued*)

Table 3-1. (*continued*)

Python function	Example

re.split( ): Splits the string at the locations where the search pattern is found in the string being searched.	CODE:
	`re.split('3','98371234')`
	Output:
	`['98', '712', '4']`
	The string is split into smaller string wherever the search pattern, “3”, is found.

re.sub( ): Substitutes the search pattern with another string or pattern.	CODE:
	`re.sub('3','three','98371234')`
	Output:
	`'98three712three4'`
	The character “3” is replaced with the string ‘three’ in the string.

Further reading:

Learn more about the functions discussed in the above table:

- Search and match function: https://docs.python.org/3.4/library/re.html#search-vs-match
- Split function: https://docs.python.org/3/library/re.html#re.split
- Sub function: https://docs.python.org/3/library/re.html#re.sub
- Findall function: https://docs.python.org/3/library/re.html#re.findall

Metacharacters

Metacharacters are characters used in regular expressions that have a special meaning. These metacharacters are explained in the following, along with examples to demonstrate their usage.

1. **Dot (.) metacharacter**

   This metacharacter matches a single character, which could be a number, alphabet, or even itself.

   In the following example, we try to match three-letter words (from the list given after the comma in the following code), starting with the two letters "ba".

   CODE:

```
re.findall("ba.","bar bat bad ba. ban")
```

   Output:

```
['bar', 'bat', 'bad', 'ba.', 'ban']
```

   Note that one of the results shown in the output, "ba.", is an instance where the . (dot) metacharacter has matched itself.

2. **Square brackets ([]) as metacharacters**

   To match any one character among a set of characters, we use square brackets ([ ]). Within these square brackets, we define a set of characters, where one of these characters must match the characters in our text.

   Let us understand this with an example. In the following example, we try to match all strings that contain the string "ash", and start with any of following characters – 'c', 'r', 'b', 'm', 'd', 'h', or 'w'.

   CODE:

```
regex=re.compile(r'[crbmdhw]ash')
regex.findall('cash rash bash mash dash hash wash crash ash')
```

   Output:

```
['cash', 'rash', 'bash', 'mash', 'dash', 'hash', 'wash', 'rash']
```

   Note that the strings "ash" and "crash" are not matched because they do not match the criterion (the string needs to start with exactly one of the characters defined within the square brackets).

3. **Question mark (?) metacharacter**

   This metacharacter is used when you need to match at most one occurrence of a character. This means that the character we are looking for could be absent in the search string or occur just once. Consider the following example, where we try to match strings starting with the characters "Austr", ending with the characters, "ia", and having zero or one occurrence of each the following characters – "a", "l", "a", "s".

   CODE:

```
regex=re.compile(r'Austr[a]?[l]?[a]?[s]?ia')
regex.findall('Austria Australia Australasia Asia')
```

   Output:

```
['Austria', 'Australia', 'Australasia']
```

   Note that the string "Asia" does not meet this criterion.

4. **Asterisk (*) metacharacter**

   This metacharacter can match zero or more occurrences of a given search pattern. In other words, the search pattern may not occur at all in the string, or it can occur any number of times.

   Let us understand this with an example, where we try to match all strings starting with the string, "abc", and followed by zero or more occurrences of the digit –"1".

   CODE:

```
re.findall("abc[1]*","abc1 abc111 abc1 abc abc111111111111 abc01")
```

   Output:

```
['abc1', 'abc111', 'abc1', 'abc', 'abc111111111111', 'abc']
```

   Note that in this step, we have combined the compilation and search of the regular expression in one single step.

5. **Backslash (\) metacharacter**

   The backslash symbol is used to indicate a character class, which is a predefined set of characters. In Table 3-2, the commonly used character classes are explained.

Table 3-2. *Character Classes*

Character Class	Characters covered
\d	Matches a digit (0–9)
\D	Matches any character that is *not* a digit
\w	Matches an alphanumeric character, which could be a lowercase letter (a–z), an uppercase letter (A–Z), or a digit (0–9)
\W	Matches any character which is *not* alphanumeric
\s	Matches any whitespace character
\S	Matches any non-whitespace character

Another usage of the backslash symbol: Escaping metacharacters

As we have seen, in regular expressions, metacharacters like . and *, have special meanings. If we want to use these characters in the literal sense, we need to "escape" them by prefixing these characters with a \(backslash) sign. For example, to search for the text W.H.O, we would need to escape the . (dot) character to prevent it from being used as a regular metacharacter.

CODE:

```
regex=re.compile(r'W\.H\.O')
regex.search('W.H.O norms')
```

Output:

```
<re.Match object; span=(0, 5), match='W.H.O'>
```

6. **Plus (+) metacharacter**

   This metacharacter matches one or more occurrences of a search pattern. The following is an example where we try to match all strings that start with at least one letter.

   CODE:

   ```
   re.findall("[a-z]+123","a123 b123 123 ab123 xyz123")
   ```

   Output:

   ```
   ['a123', 'b123', 'ab123', 'xyz123']
   ```

7. **Curly braces {} as metacharacters**

   Using the curly braces and specifying a number within these curly braces, we can specify a range or a number representing the number of repetitions of the search pattern.

   In the following example, we find out all the phone numbers in the format "xxx-xxx-xxxx" (three digits, followed by another set of three digits, and a final set of four digits, each set separated by a "-" sign).

   CODE:

   ```
   regex=re.compile(r'[\d]{3}-[\d]{3}-[\d]{4}')
   regex.findall('987-999-8888 99122222 911-911-9111')
   ```

   Output:

   ```
   ['987-999-8888', '911-911-9111']
   ```

   Only the first and third numbers in the search string (`987-999-8888, 911-911-9111`) match the pattern. The \d metacharacter represents a digit.

   If we do not have an exact figure for the number of repetitions but know the maximum and the minimum number of repetitions, we can mention the upper and lower limit within the curly braces. In the following example, we search for all strings containing a minimum of six characters and a maximum of ten characters.

CODE:

```
regex=re.compile(r'[\w]{6,10}')
regex.findall('abcd abcd1234,abc$$$$$,abcd12 abcdef')
```

Output:

```
['abcd1234', 'abcd12', 'abcdef']
```

8. **Dollar ($) metacharacter**

   This metacharacter matches a pattern if it is present at the end of the search string.

   In the following example, we use this metacharacter to check if the search string ends with a digit.

   CODE:

   ```
   re.search(r'[\d]$','aa*5')
   ```

   Output:

   ```
   <re.Match object; span=(3, 4), match="5">
   ```

   Since the string ends with a number, a match object is returned.

9. **Caret (^) metacharacter**

   The caret (^) metacharacter looks for a match at the beginning of the string.

   In the following example, we check if the search string begins with a whitespace.

   CODE:

   ```
   re.search(r'^[\s]','   a bird')
   ```

   Output:

   ```
   <re.Match object; span=(0, 1), match=' '>
   ```

Further reading: Learn more about metacharacters: https://docs.python.org/3.4/library/re.html#regular-expression-syntax

Let us now discuss another library, Sympy, which is used for solving a variety of math-based problems.

Using Sympy for math problems

SymPy is a library in Python that can be used for solving a wide range of mathematical problems. We initially look at how SymPy functions can be used in algebra - for solving equations and factorizing expressions. After this, we cover a few applications in set theory and calculus.

The SymPy module can be imported using the following statement.

CODE:

```
import sympy
```

If you have not already installed the *sympy* module, go to the Anaconda Prompt and enter the following command:

```
pip install sympy
```

Let us now use this module for various mathematical problems, beginning with the factorization of expressions.

Factorization of an algebraic expression

Factorization of expressions involves splitting them into simpler expressions or factors. Multiplying these factors gives us the original expression.

As an example, an algebraic expression, like $x^2 - y^2$, can be factorized as: (x-y)*(x+y).

SymPy provides us functions for factorizing expressions as well as expanding expressions.

An algebraic expression contains variables which are represented as “symbols” in SymPy. Before SymPy functions can be applied, a variable in Python must be converted into a symbol object, which is created using the *symbols* class (for defining multiple symbols) or the *Symbol* class (for defining a single symbol). We then import the *factor* and *expand* functions, and then pass the expression we need to factorize or expand as arguments to these functions, as shown in the following.

CODE:

```
#importing the symbol classes
from sympy import symbols,Symbol
#defining the symbol objects
```

```
x,y=symbols('x,y')
a=Symbol('a')
#importing the functions
from sympy import factor,expand
#factorizing an expression
factorized_expr=factor(x**2-y**2)
#expanding an expression
expanded_expr=expand((x-y)**3)
print("After factorizing x**2-y**2:",factorized_expr)
print("After expanding (x-y)**3:",expanded_expr)
```

Output:

```
After factorizing x**2-y**2: (x - y)*(x + y)
After expanding,(x-y)**3: x**3 - 3*x**2*y + 3*x*y**2 - y**3
```

Solving algebraic equations (for one variable)

An algebraic equation contains an expression, with a series of terms, equated to zero. Let us now solve the equation $x^2 - 5x + 6 = 0$, using the *solve* function in SymPy.

We import the *solve* function from the SymPy library and pass the equation we want to solve as an argument to this function, as shown in the following. The *dict* parameter produces the output in a structured format, but including this parameter is optional.

CODE:

```
#importing the solve function
from sympy import solve
exp=x**2-5*x+6
#using the solve function to solve an equation
solve(exp,dict=True)
```

Output:

```
[{x: 2}, {x: 3}]
```

Solving simultaneous equations (for two variables)

The *solve* function can also be used to solve two equations simultaneously, as shown in the following code block.

CODE:

```
from sympy import symbols,solve
x,y=symbols('x,y')
exp1=2*x-y+4
exp2=3*x+2*y-1
solve((exp1,exp2),dict=True)
```

Output:

```
[{x: -1, y: 2}]
```

Further reading: See more on the solve function:
https://docs.sympy.org/latest/modules/solvers/solvers.html#algebraic-equations

Solving expressions entered by the user

Instead of defining the expressions, we can have the user enter expressions using the *input* function. The issue is that the input entered by the user is treated as a string, and SymPy functions are unable to process such an input.

The *sympify* function can be used to convert any expression to a type that is compatible with SymPy. Note that the user must enter mathematical operators like *, **, and so on when the input is entered. For example, if the expression is 2*x+3, the user cannot skip the asterisk symbol while entering the input. If the user enters the input as 2x+3, an error would be produced. A code example has been provided in the following code block to demonstrate the *sympify* function.

CODE:

```
from sympy import sympify,solve
expn=input("Input an expression:")
symp_expn=sympify(expn)
solve(symp_expn,dict=True)
```

Output:

```
Input an expression:x**2-9
[{x: -3}, {x: 3}]
```

Solving simultaneous equations graphically

Algebraic equations can also be solved graphically. If the equations are plotted on a graph, the point of intersection of the two lines represents the solution.

The plot function from the *sympy.plotting* module can be used to plot the equations, with the two expressions being passed as arguments to this function.

CODE:

```
from sympy.plotting import plot
%matplotlib inline
plot(x+4,3*x)
solve((x+4-y,3*x-y),dict=True)
```

Output (shown in Figure 3-1).

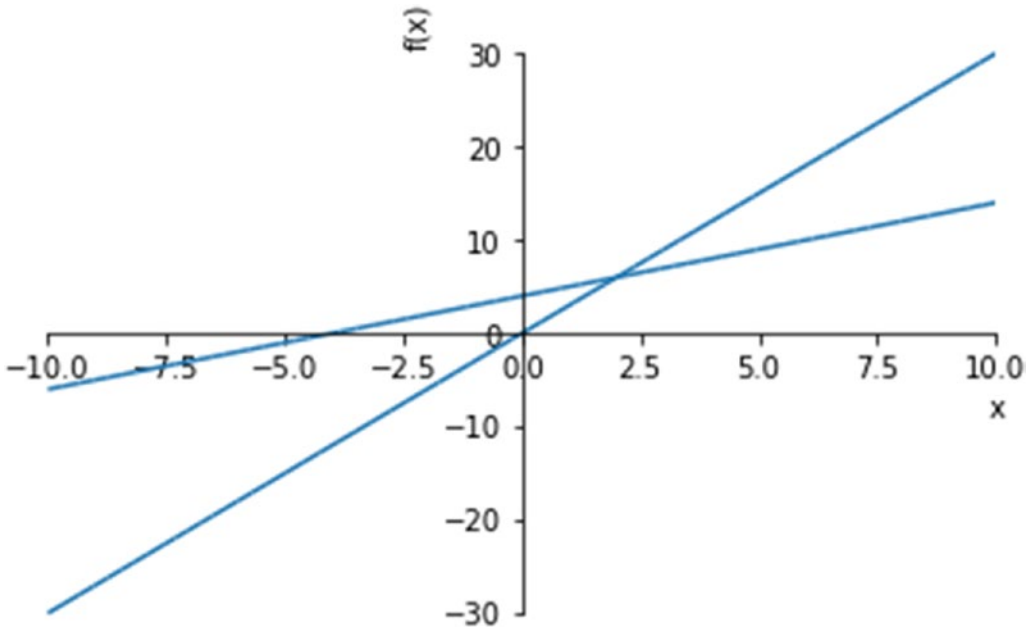

Figure 3-1. *Solving simultaneous equations with graphs*

Creating and manipulating sets

A set is a collection of unique elements, and there is a multitude of operations that can be operations that can be applied on sets. Sets are represented using Venn diagrams, which depict the relationship between two or more sets.

SymPy provides us with functions to create and manipulate sets.

First, you need to import the *FiniteSet* class from the SymPy package, to work with sets.

CODE:

```
from sympy import FiniteSet
```

Now, declare an object of this class to create a set and initialize it using the numbers you want in your set.

CODE:

```
s=FiniteSet(1,2,3)
```

Output:

```
{1,2,3}
```

We can also create a set from a list, as shown in the following statement.

CODE:

```
l=[1,2,3]
s=FiniteSet(*l)
```

Union and intersection of sets

The union of two sets is the list of all distinct elements in both the sets while the intersection of two sets includes the elements common to the two sets.

SymPy provides us a means to calculate the union and intersection of two sets using the *union* and *intersect* functions.

We use the FiniteSet class to create sets, and then apply the *union* and *intersect* functions on them, as demonstrated below.

CODE:

```
s1=FiniteSet(1,2,3)
s2=FiniteSet(2,3,4)
union_set=s1.union(s2)
intersect_set=s1.intersect(s2)
print("Union of the sets is:",union_set)
print("Intersection of the sets is:",intersect_set)
```

Output:

```
Union of the sets is: {1, 2, 3, 4}
Intersection of the sets is: {2, 3}
```

Finding the probability of an event

The probability of an event is the likelihood of the occurrence of an event, defined numerically.

Using sets to define our events and sample space, we can solve questions in probability with the help of SymPy functions.

Let us consider a simple example, where we find the probability of finding a multiple of 3 among the first ten natural numbers.

To answer this, we first define our sample space, "s", as a set with numbers from 1 to 10. Then, we define the event, denoted by the letter 'a', which is the occurrence of a multiple of 3. We then find the probability of this event ('a') by defining the number of elements in this event by the number of elements in the sample space using the *len* function. This is demonstrated in the following.

CODE:

```
s=FiniteSet(1,2,3,4,5,6,7,8,9,10)
a=FiniteSet(3,6,9)
p=len(a)/len(s)
p
```

Output:

```
0.3
```

Further reading:

See more about the operations that can be performed on sets: https://docs.sympy.org/latest/modules/sets.html#compound-sets

All information related to sets in SymPy: https://docs.sympy.org/latest/modules/sets.html#module-sympy.sets.sets

Solving questions in calculus

We will learn how to use SymPy to calculate the limiting value, derivate, and the definite and indefinite integral of a function.

Limit of a function

The limiting value of the function, f(x), is the value of the function as x approaches a particular value.

For example, if we take the function 1/x, we see that as x increases, the value of 1/x goes on reducing. As x approaches an infinitely large value, 1/x becomes closer to 0. The limiting value is calculated using the SymPy function - *limit*, as shown below.

CODE:

```
from sympy import limit,Symbol
x=Symbol('x')
limit(1/x,x,0)
```

Output:

∞

Derivative of a function

The derivative of a function defines the rate of change of this function with respect to an independent variable. If we take distance as the function and time as the independent variable, the derivate of this function is the rate of change of this function with respect to time, which is speed.

SymPy has a function, *diff*, which takes the expression (whose derivative is to be calculated) and the independent variable as arguments, and returns the derivative of the expression.

CODE:

```
from sympy import Symbol,diff
x=Symbol('x')
#defining the expression to be differentiated
expr=x**2-4
#applying the diff function to this expression
d=diff(expr,x)
d
```

Output:

$2x$

Integral of a function

The integral of a function is also called an anti-derivate. The definite integral of a function for two points, say "p" and "q", is the area under the curve between limits. For an indefinite integral, these limits are not defined.

In SymPy, the integral can be calculated using the *integrate* function.

Let us calculate the indefinite integral of the differential (2x) of the function we saw in the last example.

CODE:

```
from sympy import integrate
#applying the integrate function
integrate(d,x)
```

Output:

$x2$

Let us calculate the definite integral of the above output, using the integrate function. The arguments accepted by the integrate function include the limits, 1 and 4 (as a tuple), along with the variable (symbol), "x".

CODE:

```
integrate(d,(x,1,4))
```

Output:

```
15
```

Further reading: See more on the functions for differentiation, integration, and calculating limits: https://docs.sympy.org/latest/tutorial/calculus.html

Summary

1. A regular expression is a combination of literals and metacharacters and has a variety of applications.
2. A regular expression can be used to search and replace words, locating files in your system, and web crawling or scraping programs. It also has applications in data wrangling and cleaning operations, in validating the input of the user in email and HTML forms, and in search engines.
3. In Python, the *re* module provides support for regular expressions. The commonly used functions in Python for regular expression matching are: *findall*, *search*, *match*, *split*, and *sub*.
4. Metacharacters are characters with special significance in regular expressions. Each metacharacter has a specific purpose.
5. Character classes (beginning with a backslash symbol) are used to match a predefined character set, like numbers, alphanumeric characters, whitespace characters, and so on.
6. Sympy is a library used for solving mathematical problems. The basic building block used in Sympy is called a "symbol", which represents a variable. We can use the functions of the Sympy library to factorize or expand an expression, solve an equation, differentiate or integrate a function, and solve problems involving sets.

In the next chapter, we will learn another Python module, NumPy, which is used for creating arrays, computing statistical aggregation measures, and performing computations. The NumPy module also forms the backbone of Pandas, a popular library for data wrangling and analysis, which we will discuss in detail in Chapter 6.

Review Exercises

Question 1

Select the incorrect statement(s):

1. A metacharacter is considered a metacharacter even when used in sets
2. The . (dot/period) metacharacter is used to match any (single) character except a newline character
3. Regular expressions are case insensitive
4. Regular expressions, by default, return only the first match found
5. None of the above

Question 2

Explain some use cases for regular expressions.

Question 3

What is the purpose of escaping a metacharacter, and which character is used for this?

Question 4

What is the output of the following statement?

```
re.findall('bond\d{1,3}','bond07 bond007 Bond 07')
```

Question 5

Match the following metacharacters with their functions:

1. +	a. Matching zero or one character
2. *	b. Matching one or more characters
3. ?	c. Matching character sets
4. [ ]	d. Matching a character at the end of a search string
5. $	e. Matching zero or more characters
6. { }	f. Specifying an interval

Question 6

Match the following metacharacters (for character classes) with their functions:

1. \d	a. Matching the start or end of a word
2. \D	b. Matching anything other than a whitespace character
3. \S	c. Matching a nondigit
4. \w	d. Matching a digit
5. \b	e. Matching an alphanumeric character

Question 7

Write a program that asks the user to enter a password and validate it. The password should satisfy the following requirements:

- Length should be a minimum of six characters
- Contain at least one uppercase alphabet, one lowercase alphabet, one special character, and one digit

Question 8

Consider the two expressions y=x**2-9 and y=3*x-11.

Use SymPy functions to solve the following:

- Factorize the expression x**2-9, and list its factors
- Solve the two equations
- Plot the two equations and show the solution graphically
- Differentiate the expression x**2-9, for x=1
- Find the definite integral of the expression 3*x-11 between points x=0 and x=1

Answers

Question 1

The incorrect options are options 1 and 3.

Option 1 is incorrect because when a metacharacter is used in a set, it is not considered a metacharacter and assumes its literal meaning.

Option 3 is incorrect because regular expressions are case sensitive ("hat" is not the same as "HAT").

The other options are correct.

Question 2

Some use cases of regular expressions include

1. User input validation in HTML forms. Regular expressions can be used to check the user input and ensure that the input is entered as per the requirements for various fields in the form.
2. Web crawling and web scraping: Regular expressions are commonly used for searching for general information from websites (crawling) and for extracting certain kinds of text or data from websites (scraping), for example phone numbers and email addresses.
3. Locating files on your operating system: Using regular expressions, you can search for files on your system that have file names with the same extension or following some other pattern.

Question 3

We escape a metacharacter to use it in its literal sense. The backslash character (\) symbol precedes the metacharacter you want to escape. For instance, the symbol "*" has a special meaning in a regular expression. If you want to use this character it without the special meaning, you need to use *

Question 4

Output

```
['bond07', 'bond007']
```

Question 5

1-b; 2-e; 3-a; 4-d; 5-c; 6-f

Question 6

1-d; 2-c; 3-b; 4-e; 5-a

Question 7

CODE:

```
import re
special_characters=['$','#','@','&','^','*']
while True:
    s=input("Enter your password")
    if len(s)<6:
        print("Enter at least 6 characters in your password")
    else:
        if re.search(r'\d',s) is None:
            print("Your password should contain at least 1 digit")
        elif re.search(r'[A-Z]',s) is None:
            print("Your password should contain at least 1 uppercase letter")
        elif re.search(r'[a-z]',s) is None:
            print("Your password should contain at least 1 lowercase letter")
        elif not any(char in special_characters for char in s):
            print("Your password should contain at least 1 special character")
        else:
            print("The password you entered meets our requirements")
            break
```

Question 8

CODE:

```
from sympy import Symbol,symbols,factor,solve,diff,integrate,plot
#creating symbols
x,y=symbols('x,y')
y=x**2-9
y=3*x-11
```

```
#factorizing the expression
factor(x**2-9)
#solving two equations
solve((x**2-9-y,3*x-11-y),dict=True)
#plotting the equations to find the solution
%matplotlib inline
plot(x**2-9-y,3*x-11-y)
#differentiating at a particular point
diff(x**2-9,x).subs({x:1})
#finding the integral between two points
integrate(3*x-11,(x,0,1))
```